A PROPOSAL FOR TCD PACKAGES

original by :
C. Kitchen TCD

1.0 INTRODUCTION

This article is intended to bring together all the ideas currently in vogue regarding the concept of "packages" as an extension to TCD COMAL. It examines the reasoning behind the TCD package proposal and defines the package mechanism. By means of examples, it is shown how to use a package. In particular, I examine two flavours of packages: packages which supply the user with a set of routines that he can incorporate into his own programs and packages that run independently. The syntax of packages is formally defined and it is shown how one may create a package.

Many times in this discussion I use the word "procedure" to refer to both procedures and functions, it should be clear from the context that the distinction is not important.

2.0 THE WHYS AND WHEREFORES

Packages were proposed as an extension to COMAL as:

1. A means of encouraging modular program development. Packages are self-contained logical units of code.

2. A means of providing a sensible overlay facility. The "USE" command brings the package into memory when needed. The "DISCARD" command removes the package from memory when it is no longer needed.

3. A flexible and general means of extending the language. Packages allow one to introduce new statements and keywords into the language.

4. Introduces the "black box" approach to programming. The concept of packages, helps in the understanding of data and procedural abstraction.

5. Encourages sharing of resources among users. Packages provide utilities which can be used by different applications.

3.0 WHAT IS A PACKAGE ?

A package is a closed module whose internal workings are hidden from the package-user, and it may only communicate with the outside through parameters. It is a mechanism which provides the user with a set of logically related computational resources which can be used together to perform a common task.

Packages facilitate the provision of 'extra' language features. Examples include:

- A String manipulation package which provides functions that operate on strings.
- A "Turtlegraphics" package which provides all the procedures necessary to move a "turtle" about on a screen.
- An interactive COMAL tutorial package which guides the novice through the various stages of COMAL programming.
- A stand-alone graphics demonstration package which draws pretty pictures on the screen.
- An Editor Package which provides all the commands necessary to edit a file.

Textually a package consists of two parts:

1. The Package Interface.

   The Package Interface is the only part the user need see in order to use the package. It describes the resources made available by the package, and it specifies how the user may communicate with the package and what reactions to expect.

   It may contain a use-oriented syntactic specification of the procedures made available for use in a program. It describes the parameters, if any, that are associated with each procedure. It describes the effect on these parameters when the procedure is called and any results returned by a functional procedure. It describes any global data areas that are accessible to the calling program, and any constant definitions that effect the workings of the package and indirectly effect the user.

For an executable package, it provides all the use-documentation necessary to operate the package effectively. It describes the actions of the package, what output to expect and any commands or menus provided.

2. The Package Body.

The Package Body contains the implementation of the resources declared in the Interface. It may include declarations of local variables and routines used by the visible resources and an initialisation part which is run when the package is initially loaded into memory.

The Body of a Package can be implemented in any High Level Language, but the interface must always be written using COMAL syntax.

Before a package can be used it must first be brought into the workspace ,the following section describes how to activate and use a package.

4.0 HOW TO USE A PACKAGE

4.1 Getting Started

Before using any package, it is necessary to read the specification to find out what the package contains and how to use correctly. To read the contents of the specification type:

LIST <package name>

It is made clear in the specification how the package can be used. Packages can be used in three different ways:

1. A package that makes available procedures and functions can only be used in the context of a program. The specification contains a use-oriented syntactic description of the procedures provided. The USE command must be issued from within the program that uses the package, and the package procedures can be used in the same way as any COMAL procedure or function. They can only be interpreted after the program which uses them is executed.

For example, an arithmetic package which supports the following basic arithmetic operations: add(x,y),sub(x,y),div(x,y),mult(x,y).

2. A package that executes immediately can be used in a program or at command level. The specification describes the user-interface and the actions of the package. The user need only load the package into memory and it starts executing at once. The USE command both loads and activates the package. If the USE is issued from a program, control passes to the package and returns to the statement following the USE in the program. If it is issued at command level, control passes to the package, it does what it was designed to do and returns to the COMAL prompt.

   For example, a COMAL tutorial package used inside a language tutorial program.

3. A package that provides a set of commands can only be used at command level. The specification describes the commands it makes available. Once loaded, it is invoked by specifying a special initialisation command. Special purpose packages and system supplied packages are of this nature.

   Examples include, an Editor, the Package Builder, a Terminal Set-Up routine.

Once the user has read the specification and is satisfied that the package contains what he needs, the package can be brought into the workspace by issuing a USE command:

USE <package name>

Packages must be explicitly called into memory before they can be used, packages are removed from memory by using the DISCARD command:

DISCARD <package name>

If more than one program USEs the same package in succession, the DISCARD command need not be used until the last program is executed. Multiple USEs of the same package does not cause an error, and swapping the same package in and out of memory for successive programs is a time waster.

The next two sections describe how the package mechanism can be used at two different levels: program level and command level. The first section describes how a package can be used to provide useful routines that can be incorporated into a user's program. The next section describes how a stand-alone package can be used immediately.

4.2 Using A Package In A Program

Examples include: Cursor Control, Turtlegraphics, Mathematical Functions, String manipulation,Calcomp.

4.2.1 Using Package Procedures -

To demonstrate the use of package procedures I will sketch out the use of a simple string manipulation package. To first discover how to use this package, type:

```
> LIST string
```

The system will then list all the information you need to know in order to use this package:

```
SPEC
// COMAL PACKAGE: string                    CREATED: 17-08-83
//
EXPORT FUNC strlen#(string$)
// Returns the LENgth of STRING
EXPORT FUNC strpos#(string$,source$)
// Returns the POSition of the first occurence of the pattern
// in SOURCE found in STRING
// Returns 0 if pattern not found
EXPORT FUNC strcopy$(source$,index#,size#)
// Returns a string containing SIZE characters copied from SO
// starting at the INDEXth position in SOURCE
//
// PARAMETERS:
//
// STRING$,SOURCE$ : String Variable or Quoted String
// SIZE#, INDEX#   : Integer Variable or Value
//
ENDSPEC
```

To illustrate the use of "string" a sample program follows:

```
10 USE string
20 longstring$ := "123456789"
30 PRINT strlen#(longstring$)
40 copycat$ := "firstpart" + strcopy$(longstring$,5,5)
50 PRINT strpos#(copycat$,"6789")
60 DISCARD string
70 END
```

4.2.3 Using An Immediately Executable Package -

The following example illustrates the use of an immediately executable package in a program.

The program uses four different packages, each contains a tutorial session for a different programming language. The packages are invoked by loading them into memory, once USEd the initialisation part of the package is run and it starts executing. These interactive tutorial packages are incorporated into a main program which presents the program user with a menu, and he can select which language he chooses to receive instruction on.

```
 10 choice := 0
 20 WHILE choice < 5 DO
 40       PRINT "Welcome to the  Language Tutor"
 50       PRINT
 60       PRINT
 70       PRINT " 1 - COMAL"
 80       PRINT " 2 - PASCAL"
 90       PRINT " 3 - C"
100       PRINT " 4 - FORTRAN"
101       PRINT " 5 - Exit the Language Tutor"
102       REPEAT
110               INPUT "Please Enter your Selection":choice
111       UNTIL choice >= 1 AND choice <= 5
120       CASE choice OF
130       WHEN 1
140               USE comal
150               DISCARD comal
160       WHEN 2
170               USE pascal
180               DISCARD pascal
190       WHEN 3
200               USE clang
210               DISCARD clang
220       WHEN 4
230               USE fortran
240               DISCARD fortran
250       OTHERWISE
260               PRINT "Good-Bye"
270       ENDCASE
280 ENDWHILE
290 END
```

4.3 Using A Package At Command Level

The Language packages used in the previous example can be used immediately at command level in the same way as they were USEd in the example program. The statement:

```
>USE fortran
```

initiates the Fortran package and it starts executing.

Section Seven demonstrates the use of the Package Builder, which is a package which provides the user with commands. It is initiated at the COMAL prompt, once invoked it operates within its own environment.

5.0 RULES GOVERNING PACKAGES

5.1 Loading A Package

Packages are brought in and out of memory under the user's control.

5.1.1 Non-Existent Package -

If a user tries to use a package that does not exist a fatal run-time error results.

5.1.2 Package Already In Memory -

If a user tries to load a package that is currently in memory, the initialisation part is run again and the package variables reset.

5.1.3 Package Not In Memory -

Trying to discard a package that is not currently in memory is a harmless error. In the context of a program, the user's command is ignored. At command level a message such as: "Package not in Memory" is generated.

5.2 When To "USE" A Package

Procedural packages,that is, packages which contain an EXPORT PROC/FUNC statement can only be used in the context of a program. The system can detect such packages by locating the EXPORT statement. If the user attempts to use a procedural package at command level, he will get the

message: "This is not an immediate package, it can only be used after a program is executed".

For command packages, that is, packages which contain the statement EXPORT COMMAND, the USE only operates at command level. A run-time error occurs if a command package is used within a program.

Packages which do not contain EXPORT statements and consequently execute immediately, can be used both at command and program level.

5.3 Multiple Names In User Programs

It is not usually necessary for writers of main programs to qualify package procedure names, as they know what packages they have brought into memory and the names that each package supplies. But if more than one package is in memory and a procedure of the same name has been exported by both or if the user defines a procedure in his program of the same name as one from a package, a conflict arises. If there is a conflict of names and the calls are not qualified, an "ambiguous reference" error will result.

5.4 External Names In COMAL Packages

References to procedures from other packages must be qualified by the called procedure's package name. This is necessary since the package designer does not know what other packages will be in memory when this package is called and packages can contain routines of the same name. If the package designer fails to qualify an external reference, he will get an "undefined reference" error message when he tries to save the package.

5.5 Variables In Packages

Even it a user knows about internal variables or procedures in a package, he may not access or call them unless they are explicitly "exported" by the package. If he tries to do so, a run-time error results. Data declared in a package cannot be accessed by the user directly by name. Variables explicitly made known as parameters in the interface, can only be read or written by the interface routines.

5.6 Listing A Package

Users may list packages, by default only the package interface is listed, whether or not the body of a COMAL package may also be listed is a privilege set by the package designer. The body of a package written in a language other than COMAL may not be listed.

6.0 SYNTAX OF PACKAGES

The package command syntax has been accepted by the TCD COMAL group, I here propose the syntax for the package declaration and the qualified reference for procedures in a package.

The syntax of the package COMMANDs is:

```
<use statement>          ::=
      USE <identifier list>

<discard statement>      ::=
      DISCARD <identifier list>

<identifier list>        ::=
      <identifier> (,<identifier list> )
```

The package commands can either be issued from a program, or directly at system level. Each identifier, in addition to being syntactically valid must correspond to a physical package.

The syntax of a package DECLARATION is:

```
<package declaration>  ::=
      PACKAGE
      <specification part>
      <implementation part>
      [ <initialisation part> ]
      ENDPACK

<specification part>   ::=
      SPEC
      [ (<export statement>) ] | <eol>
      ENDSPEC

<export statement>     ::=
      EXPORT PROC <procedure list><eol> |
      EXPORT FUNC <function list><eol>
      EXPORT COMMAND <command list><eol>

<eol>                  ::=
      [<remark>]<newline>

<remark>               ::=
```

```
        // (<displayable character>)

<procedure list>        ::=
        <procedure heading> (,<procedure heading> )

<function list>         ::=
        <function heading> (,<function heading> )

<command list>          ::=
        <identifier> (,<identifier> )

<procedure heading>     ::=
        <procedure identifier><head appendix>

<function heading>      ::=
        <function identifier><head appendix>
```

NOTE that the specification must always be written using COMAL syntax, whether it be for a COMAL or a machine code package.

```
<implementation part>   ::=
        BODY
        <COMAL program> | <include statement>
        ENDBODY

<include statement>     ::=
        INCLUDE <machine code program name>

<initialisation part>   ::=
        (<non declaration statement>)
```

As many of our ideas on what a package is and how it behaves are based on the ADA language, I propose the following dot qualification in accessing procedures of a package:

```
<procedure identifier> ::=
        <package identifier>.<identifier>
```

7.0 HOW TO BUILD A PACKAGE

7.1 The Package Builder Interface

The Package-Builder (PB) is 'the' system supplied interactive package for building user-defined packages. The only way one can create ,edit or delete a package is via the PB.

The PB like other command packages operates in it's own environment,at a level above that of the COMAL environment. The PB has it's own workspace and does not infringe on the user's workspace. When I refer to workspace in this section, I am referring to the PB's workspace (ws) . To find out how to use this utility type:

```
> LIST pbuild
```

This will then list all the commands available to the PB user:

```
SPEC
// PACKAGE: pbuild       CREATED: 20-07-83
//
EXPORT COMMAND pbuild,pcreate,pcopy,pdelete,
EXPORT COMMAND pexit,pinsert,plist,pnumber
//
// To activate the Package Builder type:
//
//  PBUILD
//
// The Package Builder then indicates it's readiness to accep
// commands by displaying its prompt, PB>
// PB commands are as follows:
//
// PCREATE <package name>
//                 Used to enter package definition mode. It
//                 provides automatic line numbering facilities
//                 line-by-line syntax checking.The user escapes
//                 from package definition mode by typing ^Z. If
//                 the user fails to enter a unique package name
//                 he will be prompted for one.
//                 If there is a package already in the
//                 workspace the PB asks for confirmation
//                 before deleting it.
//
// PCOPY FROM <package name>
//                 Load the contents of <package name> into the
// PCOPY
//                 Save the contents of the ws in <package name>
//                 <package name> is the name given to the packa
//                 when it was originally created, or the name o
//                 the package that was copied into the ws.
//                 The system first checks that the package in t
//                 ws has the correct package format before savi
//                 it.
```

```
        // PDELETE
        //              Delete the current ws
        // PDELETE <range>
        //              Delete lines <range> of the ws
        // PDELETE <package name>
        //
        // PEXIT
        //              Leave Package Builder Mode and return to the
        //              COMAL environment
        //
        // PINSERT [AFTER] <linenum> <increment>
        //              Line numbers are provided to insert user-
        //              generated text, starting at <linenum> in
        //              increments of <increment>
        //
        // PINSERT <program name> [AFTER] <linenum>
        //              Pre-written COMAL programs may be inserted in
        //              the current ws, starting at <linenum>.
        //              Useful for converting a COMAL program into a
        //              package. The code of the program can be inser
        //              into the body of a package, provided there is
        //              enough space to take it.
        //
        // PLIST
        //              List the current ws
        // PLIST <range>
        //              List the lines <range> of the ws (inclusive)
        //
        // PNUMBER startline linenum increment
        //              Renumber the ws, starting at <startline>,givi
        //              it the number <linenum> and increasing each
        //              subsequent line number by <increment>
        //
        ENDSPEC

    >
```

To quickly find out what commands the package makes available, type:

```
> SHOW COMMANDS OF pbuild
```

This will list all the commands exported by the package:

```
PBUILD
PCREATE
PCOPY
PDELETE
PEXIT
PINSERT
PLIST
PNUMBER
>
```

Now that we have seen what commands the PB makes available, I shall go on to describe how one might use it to create a simple "arithmetic" package.

7.2 Using The Package Builder

To use the PB first load it into memory:

```
> USE pbuild
```

Once in memory, you can run the Package Builder, activate it by typing:

```
> pbuild
```

The system responds with the PB prompt and is ready to accept commands. To create the package "arith" type:

```
PB> pcreate arith
```

The PCREATE command acts in a manner similar to the PROGRAM command, providing automatic line numbers in increments of 10. One can then enter text and exit package definition mode by typing Z:

```
 10 PACKAGE
 20 SPEC
 30 // COMAL PACKAGE: arith                CREATED: 17-08-83
 40 // ADDition,SUBtraction,DIVision,MULTiplication of two int
 50 // MAXINT = X
 60 // MININT = X
 70 // Functions return 0 on error condition
 80 //
 90 EXPORT FUNC add$(x$,y$),sub$(x$,y$),div(x$,y$),mult$(x$,y$
100 //
110 ENDSPEC
120 BODY arith
130       FUNC add$(x$,y$)
140             z$:=x$+y$
150             RETURN(z$)
160       ENDFUNC
170       FUNC mult$(x$,y$)
160             z$:=x$ * y$
170             RETURN(z$)
180       ENDFUNC
190       FUNC div(x$,y$)
200             IF y$=0 THEN
210                   RETURN(0)
220             ELSE
230                   z := x$/y$
240                   RETURN(z)
250             ENDIF
260       ENDFUNC
270       FUNC sub$(x$,y$)
280             z$ := x$ - y$
290             RETURN(z$)
```

```
300     ENDFUNC
301     END
310 ENDBODY
320 ENDPACK
330 ^Z

PB>
```

The Body of a COMAL package has the same syntax as a COMAL program and the Line-by-Line compiler checks the syntax of each line entered. As the package-designer does not have the convenience of writing a program to test and execute a package from within the PB, the PB traps as many errors as soon as is possible. This saves the user from having to enter the PB each time he wants to fix the package.

Before saving a package, the PB does a prepass over the code, along with checking all the structures that the COMAL prepass checks, it does some checks of its own:

1. Direct recursion is not allowed, if package "arith" contains the statement: 'USE arith' or 'DISCARD arith', an error message is generated.

2. It checks that all exported routines are defined, and that functions are defined as functions and procedures as procedures.

3. It also checks that the package keywords i.e PACKAGE,SPEC,ENDSPEC... are present and in correct order. The SPEC block can only contain lines beginning with "//" or "EXPORT" and the initialisation part only contains non-declaration statements.

4. When the prepass has looked over the program it knows which references are undefined. The only undefined references it can accept are calls to the COMAL pre-defined functions (they too can be packaged). For all other external names there must be a USE statement in the program, which the prepass can check against qualified names. Non-local package procedures that are not qualified generate an "undefined reference" error.

The PB will not save the package until all syntax and prepass errors are resolved.

Before finally saving the package, the PB requests the listing privilege granted to users of the package:

```
Do you wish to allow the Body of the Package to be listed (Y/N) ? |_|
```

If the designer allows the body to be listed, it will appear after the specification when the package is listed.

The package cannot be tested from within the PB, to test, one must write a driver program that calls the procedures of the package. Run-time errors are the same as those for a COMAL program, except that the line number is preceded by the package name.

When the Package Builder is no longer needed, it should be removed from memory and the space it occupied returned to the system. It is unloaded by entering:

```
> DISCARD pbuild
```

Original by :
M. Mac An Airchinnigh TCD